Packaging System

Introduction

The current `eups/setup` package management and versioning system works fairly well, but it has some perceived deficiencies. This document attempts to lay out a complete and consistent structure for managing packages, both LSST-developed and external, during development and production deployment.

Goals

Goals that are not currently met by `eups/setup` are marked with "new". Goals that are partially met by `eups/setup` are marked with "improved".

- Simple to use (improved).
- Reliable (improved).
- Portable across platforms.
- Distinguishes between installed packages and activated packages.
- Does not allow incompatible packages to be activated without explicit override (new).
- Allows any compatible set of package versions to be activated (improved).
- Allows the set of activated packages to be reconfigured rapidly, or easily allows separate environments to be configured with different sets of activated packages.
- Allows a frozen, tested, guaranteed-compatible set of package versions to be defined and distributed (improved).
- Compatible with automated builds (improved).
- Does not require specification of indirect dependencies (improved).
- Given package dependencies, automatically provides header and library dependencies for building from source (new).

Definitions and Common Features

A *package* is a set of files in a directory structure. A package has a *version* number, which must have a total order, and a set of *dependencies*, which are the names of other packages that it directly requires to function and the version ranges of those packages that it is known to be compatible with. The version ranges may be unbounded if packages are expected to be backward-compatible. *Direct dependencies* are those packages that are explicitly used by the depending package; *indirect dependencies* are packages used by the direct dependencies that are not explicitly used by the depending package. A package also has an (optional) *interface*, composed of the names of one or more header files and zero or more shared libraries.

Package dependencies, including both names and version ranges, are specified by a file named `ups/{package_name}.table` within the package. Version ranges should be as loose as possible but should disallow any combinations known to fail.

Packages may be deployed in a variety of ways, including being built from source or being distributed as binaries. Many packages, such as database client libraries, scripting languages, algorithm libraries, etc., will be obtained from external sources.

There is an official LSST installation location where the files of *installed packages* are expected to be found. This is relative to the locations specified by the `LSST_HOME` and/or `LSST_DEVEL` environment variables.

Multiple versions of the same package may be installed at the same time in this location.

One version of each package may be marked as *current*, or preferred as the default.

Developers may be working on one or more packages in Subversion working copies, building the package files in those trees. Those working copy versions may be installed into the standard location. If so, the version number associated with such an installed working copy version is `svn{revision number}`. Such a version can be compared with normal tagged versions; the svn revision number of the tagged version will be used in the comparison.

Working copies may also be *declared* but not installed. In this case, the files in the package, including its dependency specification and interface, are not copied into the standard location but are accessed directly from the working copy directory. Multiple working copies of the same package may be declared at the same time. The version number associated with a declared package is a special string, perhaps something like `LOCAL-{username}-{sequence}`; such a version also satisfies any and all dependency specifications. A declared version may be marked current, in which case it overrides any other installed version that is marked current. A declared version for one user is not visible to other users even if they share installation locations.

Installed or declared packages may be *activated*. (In the current system, this is called *setup*. A different term is used in this document to avoid the ugliness of "unsetting up" and to avoid confusion with the standard RHEL `/usr/bin/setup` command. `setup`, `unsetup`, `activate`, and `deactivate` will be accepted as commands.) At most one version of a package at a time may be activated. Only activated package versions are visible for running executables and linking libraries.

A shortcut command for simultaneously declaring, marking current, and activating a working copy will be available. (In the current system, this is "`setup -r .`".)

Activating a specific version of a package causes its entire dependency tree, including direct and indirect dependencies, to be activated, if they have not been already. The version chosen for a dependent package is the one marked current, if it satisfies all constraints, or else the latest version that satisfies all constraints. If the `--keep` option is specified, a dependent package that has already been activated will have that version preferred over the one marked current. In addition, any already-activated packages dependent on the package being activated are checked to make sure that the version being activated meets their constraints. If a set of versions that satisfies all the constraints does not exist, an error will be reported.

Activating a package without specifying a version causes the current version of the package to be activated, or the latest version that meets the constraints of already-activated packages. The package's dependencies are then activated as described above.

A complete dependency tree of specific versions for a given package or for all activated packages may be frozen as a *snapshot*, normally stored in an ordinary text file. Activating a snapshot file causes all of the listed package versions to be activated, replacing any other activations already present, as long as the constraints of any already-activated packages are met. If those constraints aren't met, an error will be reported so that the user can manually deactivate the depending package. In any case, any packages not listed in the snapshot are unchanged. Snapshot files enable complete, tested, working sets of versions to be re-created easily. In addition, they can be used to record the state of the system during or after automated build and test operations. Ideally, a version specified in a snapshot file that is not yet installed on the current system would be retrieved from a central repository as part of the activation process.

Installed packages contain both the "loose" version ranges in their table files and a snapshot of the state of their dependencies at the time of installation. The table files are normally used during activation as described above, but the `--exact` option may be specified to use the snapshot instead. This is unlike the current system, where the snapshot is always used.

Building a package from a working copy requires that the working copy be declared and activated so that compilation, linking, and testing during the build process uses the just-built files. At the beginning of the build process, `scons` (or actually `sconsUtils`) must check to make sure that all dependent packages have been activated and that the interface files (headers and libraries) that they specify are available. It will not require any additional information aside from the table file to do this.

Implementation Proposals

The following proposals are suggestions for how implementation of the above requirements could be performed. Proposal 1 will be implemented first; Proposal 2 may be implemented at a later date.

Implementation Proposal 1

This proposal is similar to the current `eups` system, in that it uses environment variables to locate packages.

When activated, each package's root directory is placed in an environment variable named `{PACKAGE_NAME}_DIR`. Its binary, library, and Python directories are added to or replaced in `PATH`, `LD_LIBRARY_PATH`, and `PYTHONPATH`, respectively.

The `SConstruct` file that controls the build process for each package must parse the dependency table. If a needed package has not been activated or has been activated with an incompatible version, an error will result. Otherwise, the `SConstruct` will arrange for all dependent packages to have their include, library, and Python directories, if present, available for all build steps. This includes creating `-I`, `-L`, and, if necessary, `-rpath` or `-rpath-link` options for compiling and building and the equivalents for SWIG.

Implementation Proposal 2

This proposal is radically different from the current `eups` system. It uses symbolic links to make package files appear in a single set of directories.

When activated, each file of each package is linked into the corresponding position in a master LSST software tree. Directories are not linked from the installation location; instead, subdirectories in the master tree are created as necessary. Upon deactivation, the links are removed; subdirectories may be removed if empty. This process guarantees that there are no hidden name collisions between the files in the packages.

The `PATH`, `LD_LIBRARY_PATH`, and `PYTHONPATH` environment variables point to this master tree and are thenceforth unchanged. The LSST-specific Python "import hook" would not be needed. If a developer needs access to multiple package sets at the same time, different shells can be configured with different master tree locations.

The `SConstruct` file need not know about package dependencies at all. It merely points the `-I`, `-L`, `-rpath`, etc. options to the appropriate locations in the master tree.

Comments

Comment by rowen on Tue 29 Apr 2008 02:30:09 PM CDT

A minor request: eliminate the need to supply the package name in "eups declare" commands. eups already knows the only valid package name and fails if it's wrong, so why specify it?

Comment by rhl on Mon 05 May 2008 02:00:13 PM CDT

I believe that Russell's `eups declare` request is already implemented

Comment by rhl on Mon 05 May 2008 02:05:40 PM CDT

Rather than add another set of files specifying "snapshots", I'd rather modify the current `eups expandtable` command to include both the relative expression (`> 1.2.3`) and the version that is actually in use (`1.2.10`). Then add an option `--exact` to setup; without setup the best version will be chosen as described above, but with `--exact` the current behavior would remain, with the change that any conflicting versions will become an error.

Reply by ktl

RHL's proposal above would give each installed package the equivalent of a snapshot taken at the time of installation. This snapshot could then be optionally used.

In addition, `eups expandtable {tablefile} > snapshot` can apparently currently be used to generate snapshots as specified above, although it would be slightly nicer to have the syntax be `eups expandtable {package}`.

Further reply by ktl

The above proposal has been incorporated into the text.

Comment by rhl on Mon 05 May 2008 02:10:05 PM CDT

I'm not sure that I like the proposed change to leave untouched any versions that are already setup. With the present system, when you say `setup foo` you'll get a deterministic setup of all current versions (providing that they satisfy the conditions in the table files).

With the proposed change this is no longer true, and we need to think through the consequences.

Note that there is a `setup --keep` option to keep any currently setup local versions (i.e. not installed anywhere on `EUPS_PATH`). We could change this option to request K-T's proposal that all currently setup products be accepted.

Reply by ktl

The principle here is to modify the user's configuration as little as possible. If a currently setup/activated version is adequate, then continue to use it. This makes it possible to manually create a set of versions from the bottom up, rather than having to start with the highest-level package and reconfigure its dependencies in downwards order.

It may be the case that the default should be to upgrade with `--keep` as the alternate, rather than making (an extended) `--keep` the default. In that case, the user must be informed when already-setup/activated package versions are being changed (likely with a better message than the current "You setup package ver1, and are now setting up ver2").

Further reply by ktl

The default has been changed to prefer current versions.

Comment by rhl on Mon 05 May 2008 02:22:26 PM CDT

K-T writes:

> If so, the version number associated with such an installed working copy version is that of the last tagged version on the same branch with +svn{revision number} appended.

Is this a proposal that we map the svn revision number to the versions that are currently tagged? This is not trivial. We'd have to search back through svn's revision history looking for an ancestor that had been tagged --- but what if the code had been branched since it was last tagged? In that case we'd have to go back past the `svn copy` looking for a tagname, but would it mean anything?

The current code names versions `svn####` which (as K-T well knows) breaks relative expressions such as > `1.2.3` --- however, its pretty easy to translate `1.2.3` back to an svn revision number allowing the relative expression to be evaluated. Would this be good enough?

Reply by ktl

Yes, the proposal was to map the svn revision number to the last-tagged version on its branch. If comparisons between svn revisions and package version numbers can be made to work, that would be adequate.

Further reply by ktl

svn revisions and declared revisions now always satisfy constraints, so they do not need to be ordered. This has deficiencies in that an svn revision from a ticket branch working on a previously-tagged version may not actually satisfy the same constraints as a ticket branch working on a trunk modification, but for now it seems too difficult to distinguish between these.

Comment by rhl on Mon 05 May 2008 02:23:31 PM CDT

The make-a-forest-of-links proposal 2 is merely an implementation detail. It could be done, but do we need it? It's certainly orthogonal to proposal 1.

Reply by ktl

Proposal 2 changes the `SCons*` files substantially. It is not clear to me whether the `+svnlatest` proposal works with the link forest; linking from a developer directory would only be possible after building the package and would not include any new files incorporated into the package during development.

Further reply by ktl

The proposals have been rewritten as pure implementation details, with Proposal 1 implemented first and Proposal 2 deferred until later.

Comment by robyn on Mon 05 May 2008 02:41:22 PM CDT

I like the simplicity of Prop2 but like Prop1's the ability to automatically download and isntall a needed package from the SVN repository.

Seems to me that a hybrid of Prop1 and Prop2 is possible. Taking Prop2 as the base, if an uninstalled but tagged package is activated, it can be auto installed in the system area and then the appropriate links built. I don't know how the issue should be resolved if that package should be from the 'trunk' -- multiple packages named '<tag>+SVNLATEST' sounds like a recipe for disaster.

Reply by ktl

I didn't mean to imply that Proposal 1 would automatically download anything from SVN. It was just trying to describe what currently happens, with slight extensions to handle dependencies in table files and to make version numbers comparable when setting up/activating directly from svn working copies (`+svnlatest`).

Uninstalled packages that are required would be obtained from the package repository, not SVN. This could be done with either proposal, although it isn't done currently.

I would generally prefer banning working copy setups/activations altogether.

Comment by rhl on Mon 05 May 2008 04:54:39 PM CDT

K-T writes:

> I would generally prefer banning working copy setups/activations altogether

I don't understand this. Saying `setup -r .` is something that I do all the time; do you mean declaring working copies? That's pretty useful too, although a private metapackage with lots of `setupRequired(foo -r $HOME/LSST/fw)` works too (I think that `$HOME` works; maybe you need `${HOME}`?)

Reply by ktl

Yup, in my ideal world, you'd have to install your just-built package even to test it. Of course, installations should be cheap and non-conflicting (due to svn revision numbers). One advantage of doing this is that testing from the installed environment is consistent with the way the package will be used, as opposed to testing in the working copy directory, which may have arbitrary stuff left around inside it.

On the other hand, I recognize that this may be onerous for developers and may slow down the build/test/debug cycle, so we probably need an exception for the package under development. This would presumably include the current working directory, so something like `setup -r .` might be appropriate. Generically allowing any directory to be setup/activated seems dangerous, however. Given that, it is not necessary to allow declarations outside the installation tree.

Further reply by ktl

Despite my misgivings, declaring and activating working copies has been explicitly added to the text.

Comment by rowen on Tue 06 May 2008 01:52:32 PM CDT

Two minor requests:

- Support wildcards in "eups list", e.g. "eups list daf*"
- Always list package names for "eups list" even if you don't use wildcards.

See Also

- Russell Owen's svn-related Setup Use Cases
- Ray Plante's Setup Use Cases
- KTL's Suggested Implementation of Software Environment Use Cases

Add comment

Your email or username: